Számítógépes Grafika

7. előadás: grafikus szerelőszalag, lokális illumináció

Bán Róbert
rob.ban@inf.elte.hu

Eötvös Loránd Tudományegyetem
Informatikai Kar

2025-2026. tavaszi félév

Tartalom

Emlékeztető

- Múlt órán megismerkedtünk a rekurzív sugárkövetéssel
- Előnyei:
  - A színtér benépesítésére minden használható, ami metszhető sugárral
  - Rekurzióval könnyen implementálható programmal
  - A fényt részecskeként kezeli, az ebből következő hatások könnyen megjeleníthetőek
- Ugyanakkor láttuk, hogy vannak hátrányai is:
  - Minden pixelnél minden primitívvel kellett tesztelnünk $\rightarrow$ ezen próbáltunk gyorsítani (dobozolás, térfelosztás)
  - Globális jellegű az algoritmus, nehezen gyorsítható hardveresen
  - A fény hullám természetéből adódó jelenségeket nem tudja visszaadni
  - Valósidejű alkalmazásokhoz túl lassú

Raycasting

```
Minden pixelre a képernyőn:
    Minden objektumra a színtérben:
        Eltalálja az objektumot a pixel sugara?
```

Inkrementális képszintézis

```
Minden objektumra a színtérben:
    Minden pixelre a képernyőn:
        A pixelt lefedi az objektum vetülete?
```

Valósidejű grafika

- Sugárkövetésnél tehát ez volt: $\forall$ pixelre indítsunk sugarat: $\forall$ objektummal (geometriával) nézzük van-e metszés
- Ehelyett próbáljuk meg ezt: $\forall$ objektumra (geometriára): számoljuk ki, mely pixelekre képeződik le és végül csak a legközelebbit jelenítsük meg!
- A sebesség nagyban függ attól, hogy milyen gyorsan dönthető el egy pixelről, hogy lefedi-e az objektum vagy sem
- Emiatt az objektumokat csak egyszerű geometriákból építhetjük fel $\Rightarrow$ ez a gyakorlatban lineáris elemeket jelent (szakaszok és háromszögek)
- Minden más geometriát (például gömb) ezekkel a *primitív geometriákkal* közelítjük, **tesszelláljuk**

Valósidejű grafika – ötletek

- Ne számoljunk feleslegesen sem: amint lehet, szűrjük ki azokat a geometriákat, amelyek biztosan nem képeződnek le a képernyőre
- Ezen kívül minden műveletet olyan koordináta-rendszerben végezzünk el, amiben a legkönnyebb végigszámolni
- A korábbi számításaink eredményeit pedig használjuk fel, ahol csak tudjuk

Inkrementális képszintézis – fogalmak

- *Koherencia*: Pixelek helyett nagyobb logikai egységekből, *primitívekből* indulunk ki
- Pontosság: *objektum tér pontosság* („pixel pontosság" helyett)
- *Vágás*: képernyőről (látótérből) kilógó elemeket távolítsuk el, ne számoljunk velük fölöslegesen
- *Inkrementális elv*: az árnyalási és takarási feladatnál kihasználjuk a nagyobb egységenként szerzett információkat (például a háromszögek meredekségét $(\partial_x z, \partial_y z)$ a fragmentek mélységének kiszámítására).

Összehasonlítás

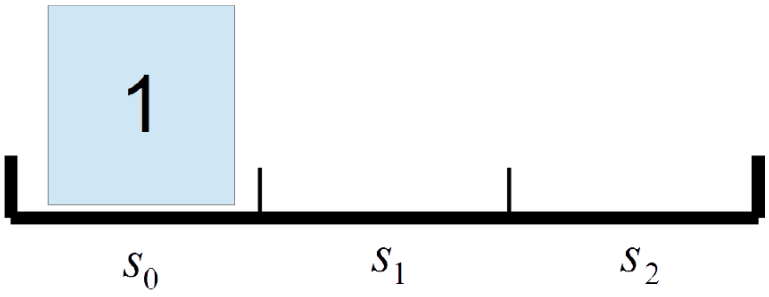
Sugárkövetés

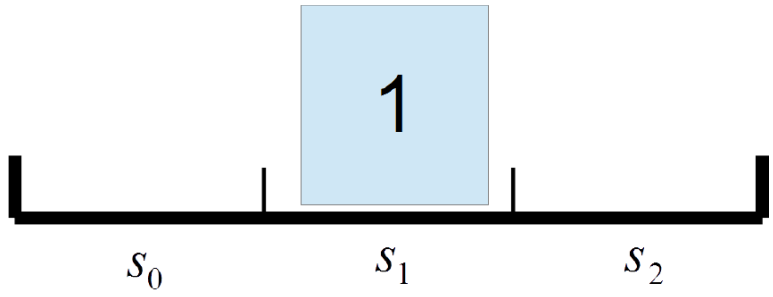
- pixelenként számol
- amit lehet sugárral metszeni, az használható
- van tükröződés, fénytörés, vetett árnyékok
- takarási feladat triviális
- sok pixel, sok sugár miatt nagy számításigény

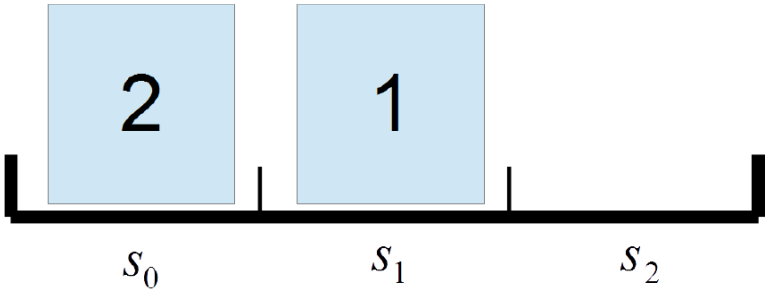
Inkrementális képszintézis

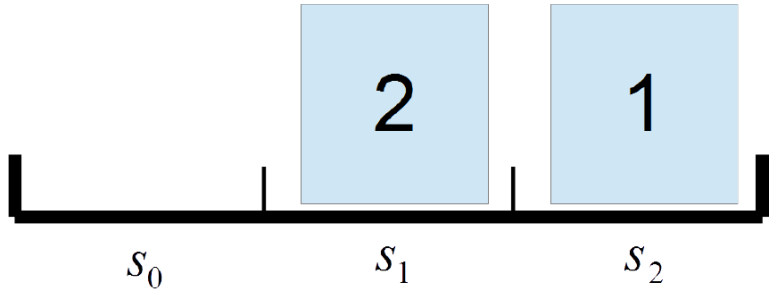
- primitívenként számol
- ami nem primitív, azt azzal kell közelíteni
- külön algoritmus kell ezekhez
- külön meg kell oldani
- a koherencia miatt kisebb számításigény

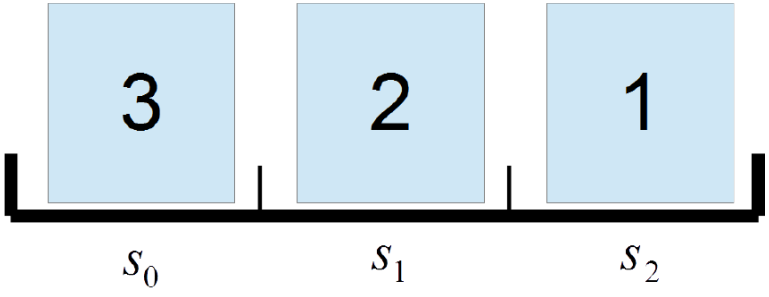
Pipeline











Pipeline

- A pipeline (vagy **szerelőszalag**, esetleg *csővezeték*) feldolgozó egységek lánca
- Az s_i feldolgozóegység bemenete az s_{i-1} feldolgozóegység kimenete
- Az s_i kimenete pedig az s_{i+1} bemenete
- Ha egy problémát fel tudunk osztani n db egymást követő részfeladatra, amely részfeladatok nagyjából ugyanannyi idő alatt elvégezhetőek, akkor egységnyi idő alatt egyszerre n munkadarabon tudunk dolgozni
- A kezdeti felfutás és az utolsó munkadarab szalagra helyezésével induló végső kifutást leszámítva

Grafikus szerelőszalag

- A színterünkről készített kép elkészítésének (szerelőszalagba szervezett) műveletsorozatát nevezik **grafikus szerelőszalagnak** (angolul *graphics pipeline*)
- A valósidejű alkalmazásoknak lényegében a színterünk leírását kell csak átadnia, a képszintézis lépéseit a grafikus szerelőszalag végzi
- A szerelőszalagban több koordinátarendszer-váltás is történik – mindent feladatot a hozzá legjobban illeszkedő rendszerben próbálunk elvégezni

A szerelőszalag működése

- A szerelőszalag minden primitív kirajzolását indító parancsra elindul (pl. glDrawArrays)
- A kirajzolandó primitívek egymástól függetlenül (általában egymással párhuzamos) mennek végig a szerelőszalag összes lépésén
- Többféleképpen megadható és csoportosítható

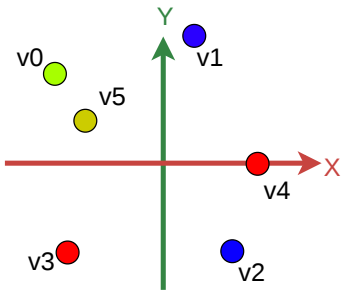
Grafikus szerelőszalag

Grafikus szerelőszalag

Grafikus szerelőszalag

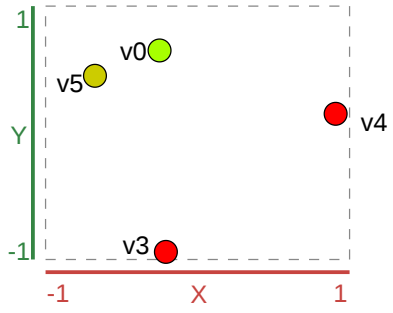
- Lépései főbb műveletek szerint:
  - Transzformációk
  - Primitívek összeszerelése
  - Vágás
  - (Homogén osztás)
  - Raszterizáció és interpoláció
  - Fragment feldolgozás
  - Megjelenítés (takarás kezelése)
- A grafikus szerelőszalag eredménye egy kép (egy kétdimenziós pixeltömb, aminek minden elemében egy színérték található)

Grafikus szerelőszalag



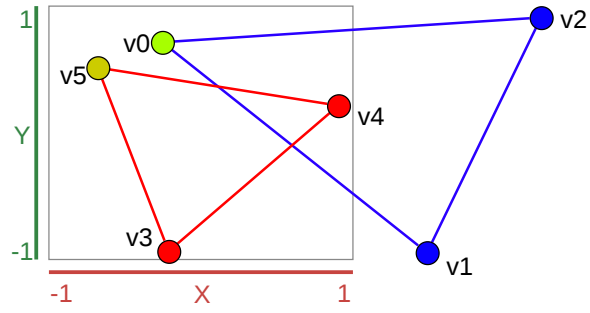
Modell k.r.
Vertexek

Transzformációk



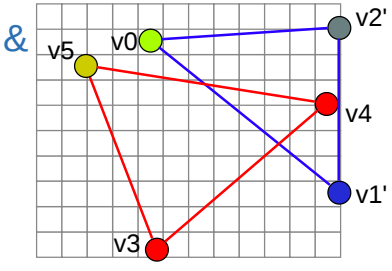
NDC
Vertexek

Primitívek összeszerelése



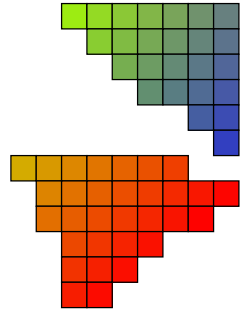
NDC
Primitívek

Vágás



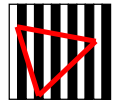
Vágott primitívek

Raszterizáció & Interpoláció

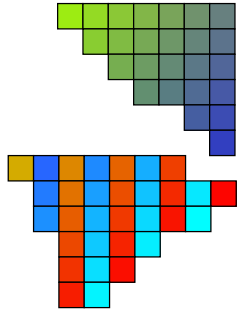


Fragmentek
(interpolált tulajdonságokkal)

Fragment feldolgozás

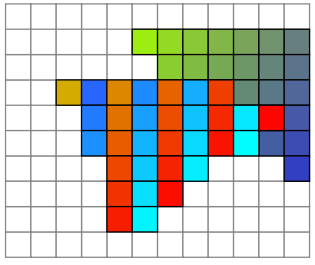


Textúra



Színezett fragmentek

Takarás



Pixelek

A szerelőszalag bemeneti adatai

- Ábrázolandó tárgyak *geometriai* és *optikai modellje*
- *Virtuális kamera* adatai (nézőpont és látógúla)
- A képkeret (az a pixeltömb, amire a színterünk síkvetületét leképezzük)
- A színtérben található fényforrásokhoz és anyagokhoz tartozó *megvilágítási adatok*

Transzformációk

- A szerelőszalag transzformációinak feladata: a modelltérben adott objektumot „eljuttatni” a képernyő-térbe
- Az objektum csúcsait (*vertex*ek) transzformáljuk
- Lépései (összesítve):

modell k.r.
→ világ k.r.
→ kamera k.r.
→ normalizált eszköz k.r. (NDC)
→ képernyő k.r.

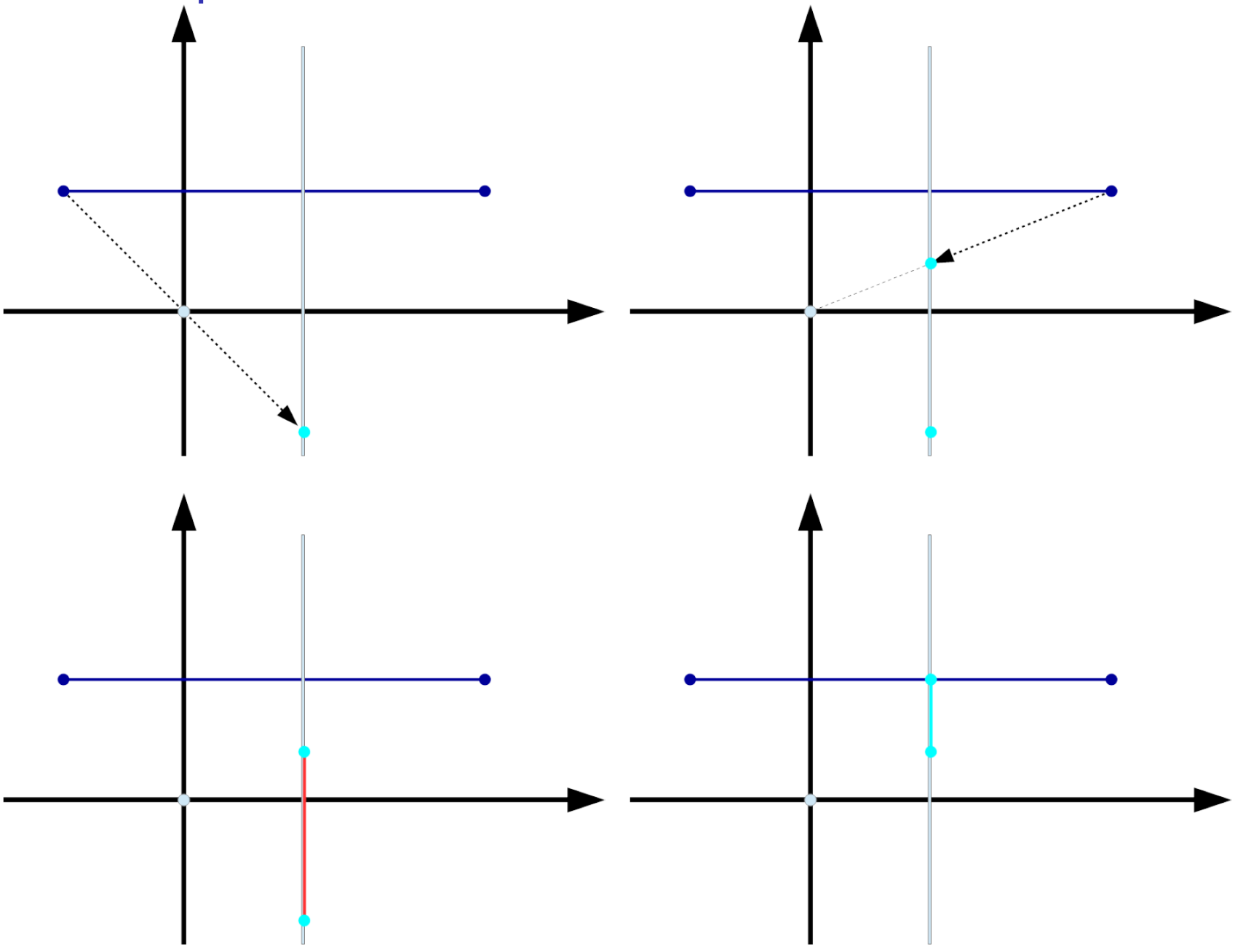
Transzformációk

- A primitívek olyanok, hogy elegendő (*) a csúcspontjaikat transzformálni és utána a transzformált csúcspontokat összekötni
- Ezért a transzformációkat a primitívek csúcspontjain végezzük el és a transzformált pontokból kötjük össze a primitíveket
- (*): ez középpontos vetítésnél óvatosan kezelendő: egyrészt vigyázni kell az átfordulásra, továbbá a baricentrikus koordináták számításánál (lásd később a perspektivikusan korrekt textúrázást, 9. előadás)

Transzformációk

- A vágás előtt nem végzünk homogén osztást, ezzel részben elkerüljük az átfordulási problémát
- Tehát a Transzformációk nevű fázisa a szerelőszalagnak nem megy végig a képernyő KR-ig, megáll a homogén osztás **előtti**, de középpontos vetítés utáni homogén térben (NDC)
- Gyakorlaton ezt írjuk bele a vertex shaderben a gl_Position változóba

Átfordulási probléma



Koordináta-rendszerek

- **Modell KR**:
  A tárolt geometria saját koordináta-rendszere. Az objektum általában az origó közelében, jól ismert helyen és irányban helyezkedik el.
- **Világ KR**:
  A modellezett virtuális tér koordináta-rendszere. Ide helyezzük el az objektumokat, itt számítjuk majd a fényeket.
- **Kamera KR**:
  A kamerához kötött koordináta-rendszer. Origója a kamera középpontja és a kamera a $-Z$ irányba néz (X jobbra és Y felfelé).

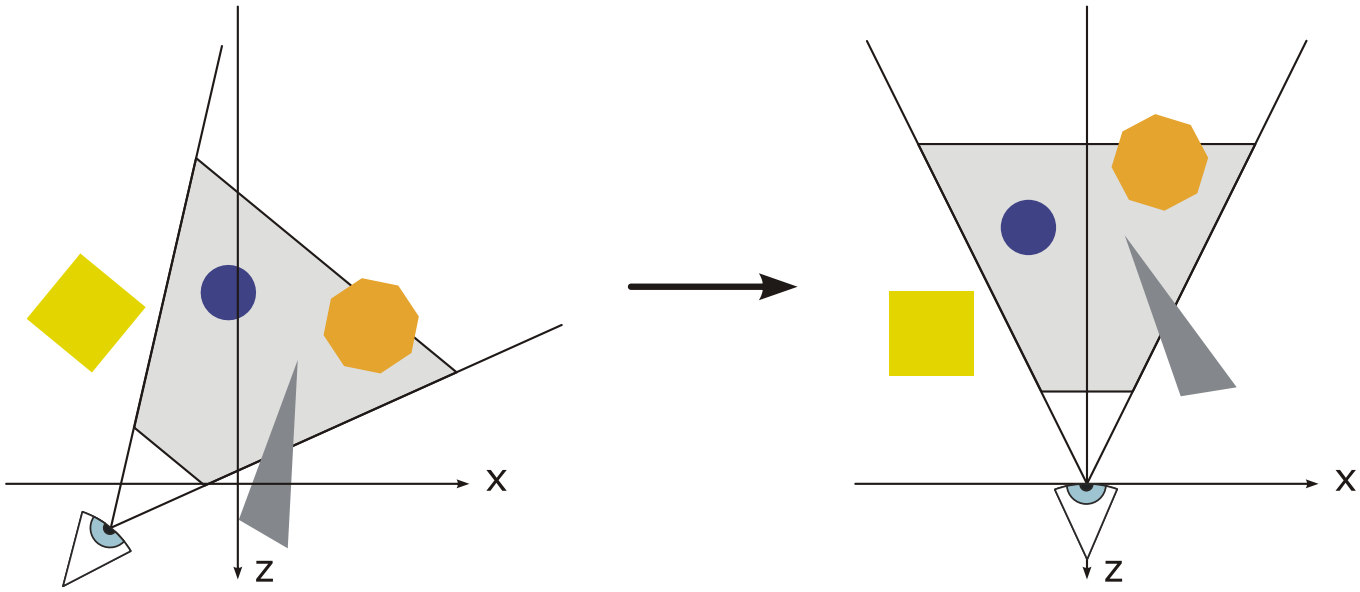
Koordináta-rendszerek

- **Normalizált eszköz KR** (*Normalized Device Coordinates – NDC*):
  Az API-ra jellemző, $[-1,1] \times [-1,1] \times [-1,1]$ (OpenGL) vagy $[-1,1] \times [-1,1] \times [0,1]$ (DirectX) kiterjedésű KR.
  A rajzterületnek felel meg (képernyő/ablak, vagy annak egy része), de független a felbontástól és képaránytól.
- **Képernyő KR**:
  A megjelenítendő képnek (képernyő/ablak) megfelelő KR (balkezes, bal-felső „sarok” az origó). Pixel koordináták.

Modellezési (világ) transzformáció

- A saját (modell) koordináta-rendszerben adott modelleket a világ-koordináta rendszerben helyezi el
- Tipikusan minden modellre különböző (lehet a színterünk két eleme csak a világtrafóban különbözik!)
- Jellemzően affin transzformációk
- Gyakorlatban: ez a *model* (vagy *world*) mátrix a kódjainkban

Nézeti (kamera) transzformáció



- A világ koordináta-rendszert a kamerához rögzített koordináta-rendszerbe viszi át.
- A transzformáció a kamera azon tulajdonságaiból adódik, melyek meghatározzák a kamera-koordinátarendszert.
- Gyakorlatban: ez a *view* vagy *camera* mátrix.

Nézeti (kamera) transzformáció

- Tulajdonságok ugyanazok, mint a sugárkövetés esetén: **eye**, **center**, **up**
- Ebből kapjuk a nézeti koordináta-rendszer tengelyeit:

$$\mathbf{w} = \frac{\mathbf{eye} - \mathbf{center}}{|\mathbf{eye} - \mathbf{center}|}$$

$$\mathbf{u} = \frac{\mathbf{up} \times \mathbf{w}}{|\mathbf{up} \times \mathbf{w}|}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

Nézeti (kamera) transzformáció mátrixa

- Áttérés az **eye** origójú, $\mathbf{u}, \mathbf{v}, \mathbf{w}$ koordináta-rendszerbe:

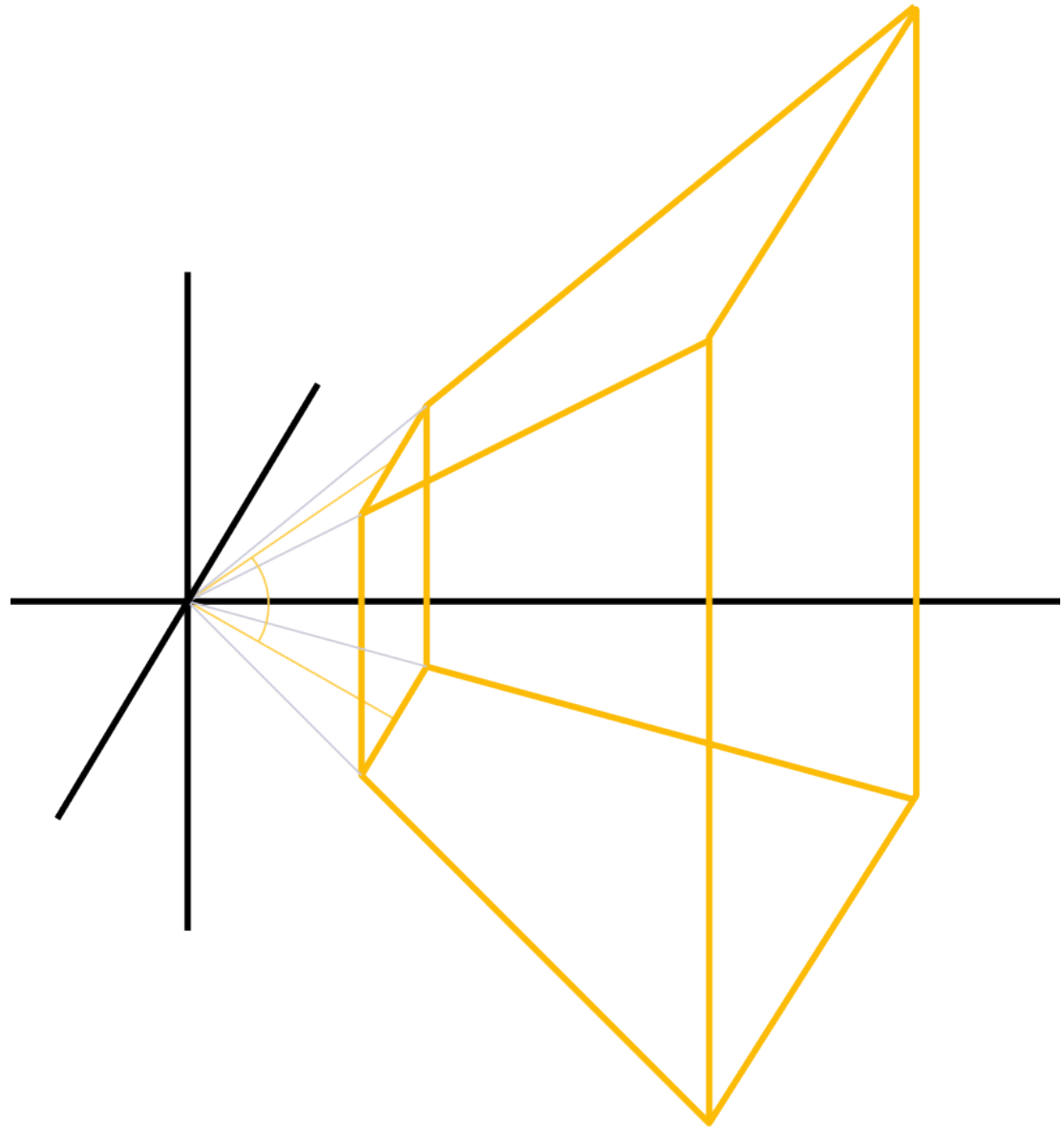
$$T_{View} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Vetítés – párhuzamos vetítés

- A mátrix ami megadja egyszerű, például az XY síkra való vetítés

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Vetítés – középpontos vetítés



Vetítés – középpontos vetítés

- Emlékeztető: 3. EA
- A nézeti csonka gúla által határolt térrészt normalizált eszköz KR-be viszi át
- Ami benne volt a csonka gúlában, az lesz benne a $[-1,1] \times [-1,1] \times [0,1]$ (vagy $[-1,1] \times [-1,1] \times [-1,1]$) tartományban
- A transzformáció a kamerán átmenő *vetítő sugarakból* párhuzamosokat csinál
- A transzformáció a kamerapozíciót a végtelenbe tolja
- Gyakorlatban: ez a *proj* mátrix

Projektív transzformáció megadása

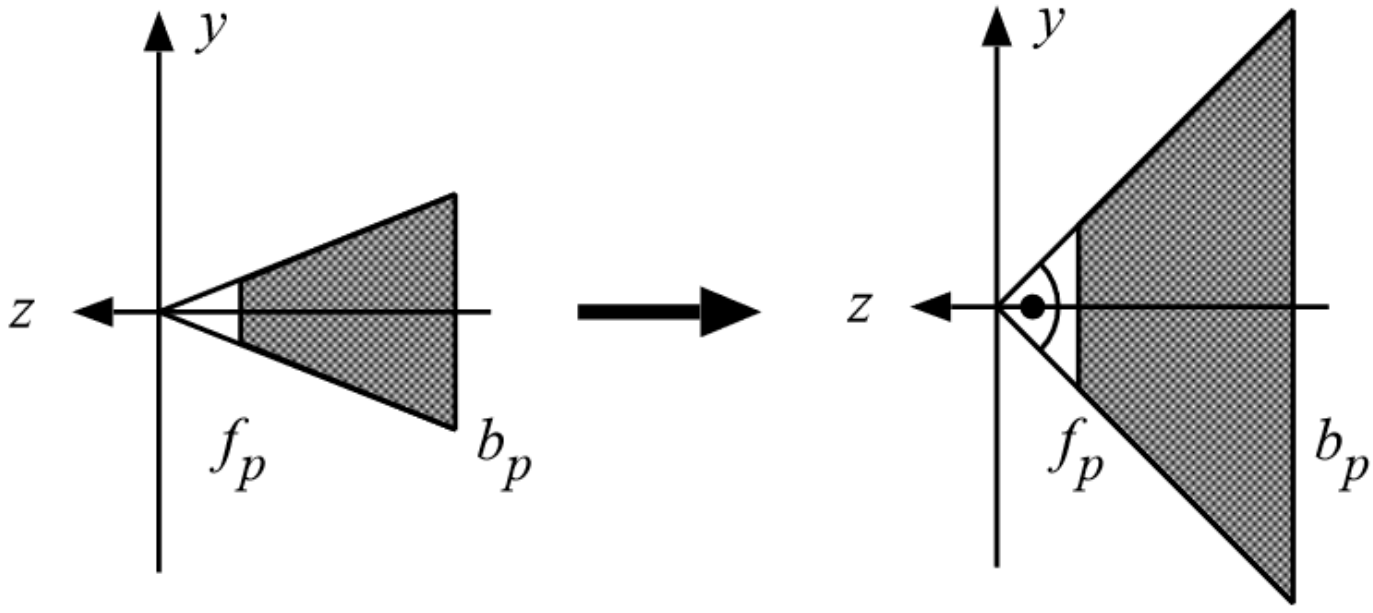
- Emlékeztető: tulajdonságok
  - függőleges és vízszintes nyílásszög *(fovx, fovy)* vagy az alap oldalainak az aránya és a függőleges nyílásszög *(fovy, aspect)*,
  - a közeli vágósík távolsága *(near)*,
  - a távoli vágósík távolsága *(far)*

Projektív transzformáció a szerelőszalagon

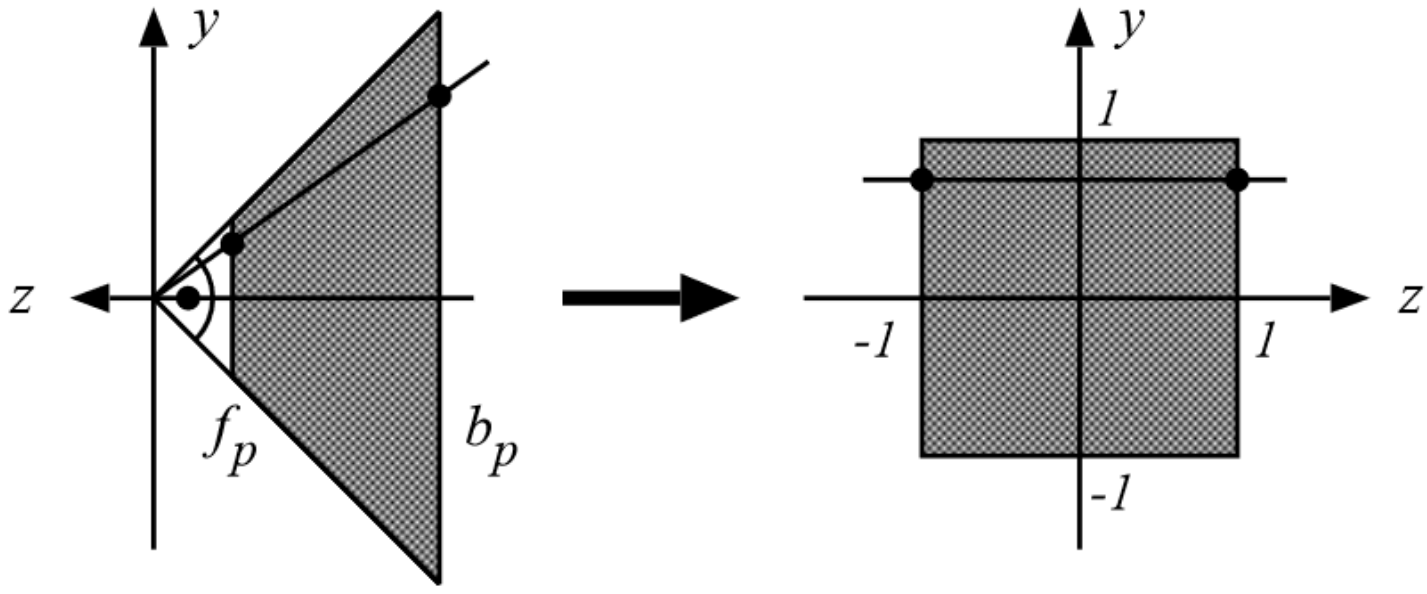
1. Normalizáljuk a látógúlát, hogy mindkét nyílásszöge $\frac{\pi}{2}$ legyen
2. Végezzük el a középpontos vetítést (az origóból, az origótól 1 távolságra lévő, XY síkkal párhuzamos síkra)
3. Képezzük át rendre a $z = near$, $z = far$ síkokat a $z = -1$ és $z = 1$ síkokra

Így (1)–(2)-vel az eredmény koordinátákat x és y szerint normalizáljuk (képezzük bele $[-1, 1]$-be), a (3)-mal pedig a z szerint

Látógúla normalizálása (1)



Látógúla normalizálása (2-3)



Látógúla normalizálása (1)

- Figyeljünk: a transzformáció ezen pontján a kamera $-Z$ felé néz és az origóban van
- A fenti térből térjünk át egy „normalizáltabb” gúlába – aminek a nyílásszöge x és y mentén is 90 fokos
- Mátrix alakban:

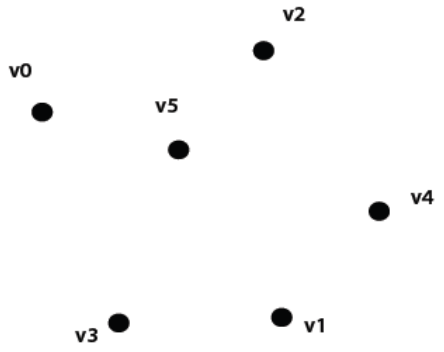
$$\begin{bmatrix} 1/\tan\frac{fovx}{2} & 0 & 0 & 0 \\ 0 & 1/\tan\frac{fovy}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

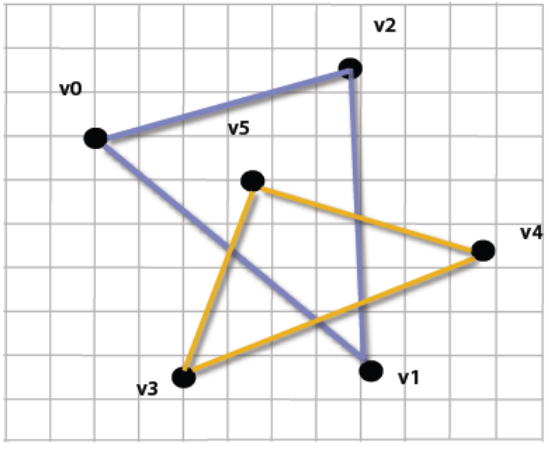
Mélység normalizálás (2-3)

- Ezután már csak a közeli és a távoli vágósík z koordinátáit kell a normalizálásnak megfelelően átképezni ($-1, 1$ vagy $0, 1$-re):

$$T_{Projection} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

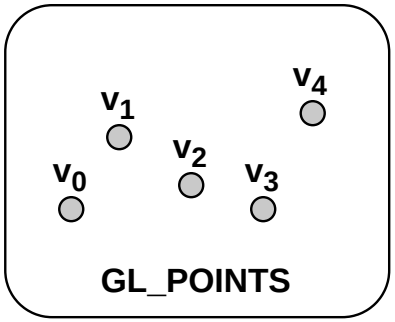
Primitívek összeszerelése

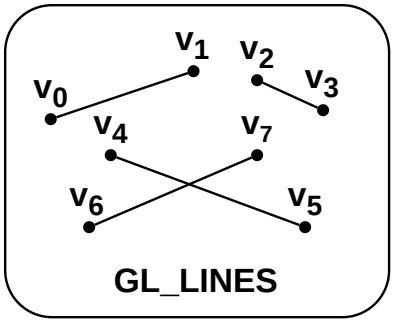


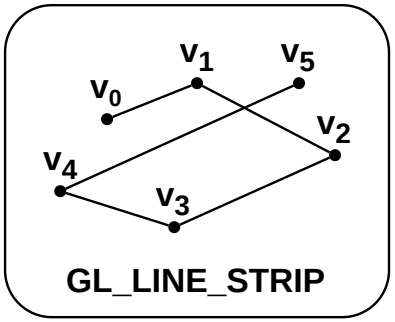


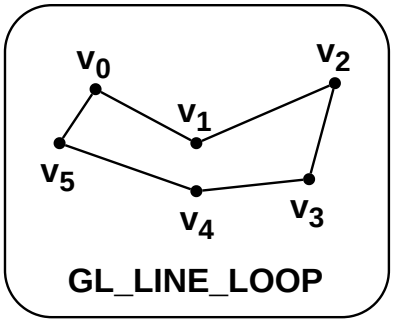
- A képernyőn (normalizált eszköz koordináta-rendszerben) elhelyezett csúcsokat (vertexek) összekötjük primitívekké
- Pont, szakasz és háromszög primitíveket különböztetünk meg
- További variációk aszerint, hogy pontosan melyik csúcsokat kötjük össze

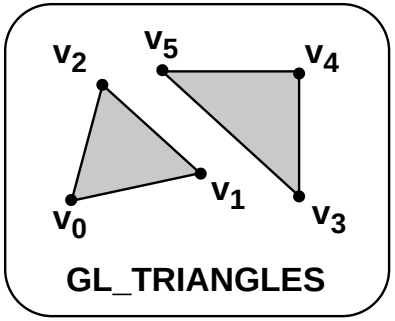
OpenGL Primitívek

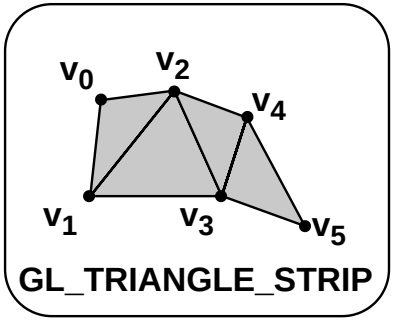


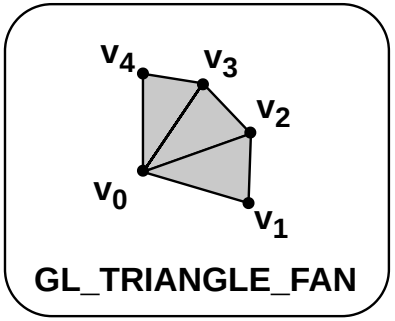




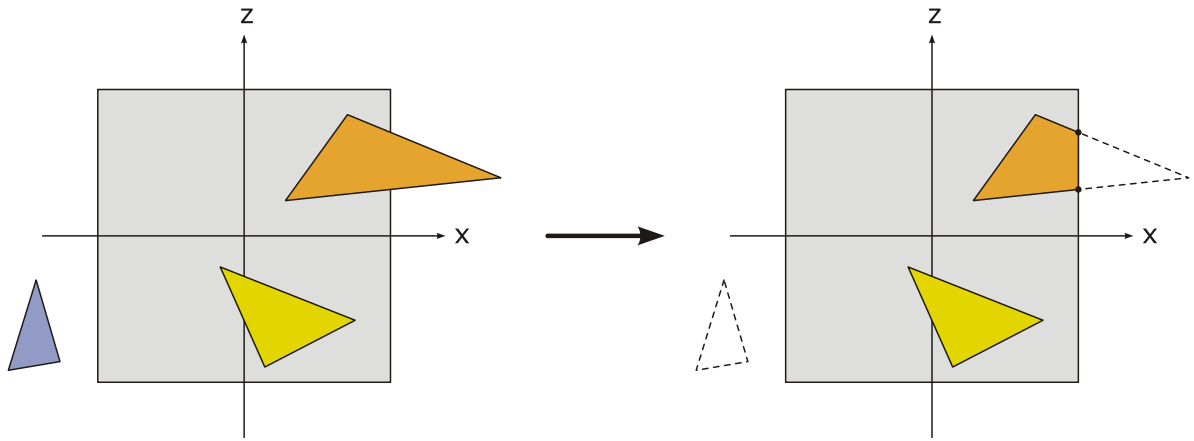








Vágás



- Primitívenként dolgozunk
- Szűrjük ki azokat a primitíveket, amik *biztosan* nem kerülhetnek a képernyőre
- Továbbá azokat, amiknek csak darabjai kerülhetnek a képernyőre *vágjuk* (azonosítsuk azon darabjait, amik teljes egészükben belelógnak a képernyőbe és fedjük le ezeket a darabokat primitívekkel)
- Cél: ne dolgozzunk feleslegesen azokkal az elemekkel, amikből nem lesz pixel (nem jelennek majd meg)

Vágás homogén koordinátákban

- Most csak egyetlen pont vágását tekintjük (szakasz és poligon később), homogén koordinátákban:
- Legyen: $[x_h, y_h, z_h, h]^T = \mathbf{M}_{\text{Proj}} \cdot [x_c, y_c, z_c, 1]^T$
- Cél:
  $[x, y, z]^T := [x_h/h, y_h/h, z_h/h]^T \in [-1, 1] \times [-1, 1] \times [-1, 1]$,
  azaz

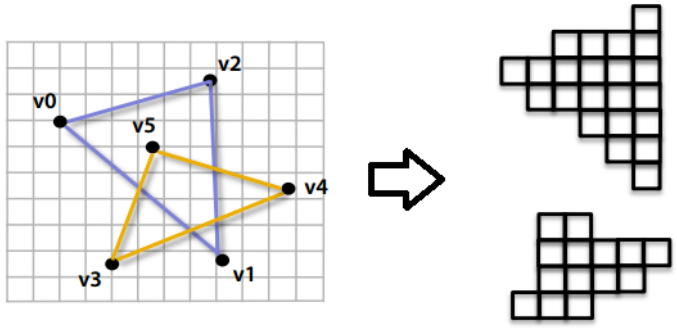
Legyen $h > 0$, és

$$-1 \leq x \leq 1$$
$$-1 \leq y \leq 1$$
$$-1 \leq z \leq 1$$

Ebből kapjuk:

$$-h \leq x_h \leq h$$
$$-h \leq y_h \leq h$$
$$-h \leq z_h \leq h$$

Raszterizáció



- Ne feledjük: eddig minden primitív, amiről beszéltünk folytonos objektum volt
- Ebben a lépésben meghatározzuk, hogy a képernyő mely pontjait (pixeleit) fedik le a primitívek
- Vagyis diszkretizáljuk a folytonos primitíveinket – ezt hívják *raszterizációnak*
- Ezzel együtt történik az interpoláció: a csúcsokban meghatározott tulajdonságokat (attribútumok) kiszámítjuk (interpoláljuk) minden egyes pixelre (fragmentre)

Raszterizáció – Miért háromszögek?

- Olyan geometriai primitíveket kell választanunk, amelyeket gyorsan tudunk raszterizálni
- Az egyik pixelhez tartozó felületi pontjának koordinátái alapján könnyen számítható a szomszédos pixelekhez tartozó pontok koordinátái
- Minden egyéb felületet ilyen primitívekkel (lényegében: síklapokkal) közelítünk ← *tesszelláció* – az objektumot eleve így adjuk meg háromszögekből

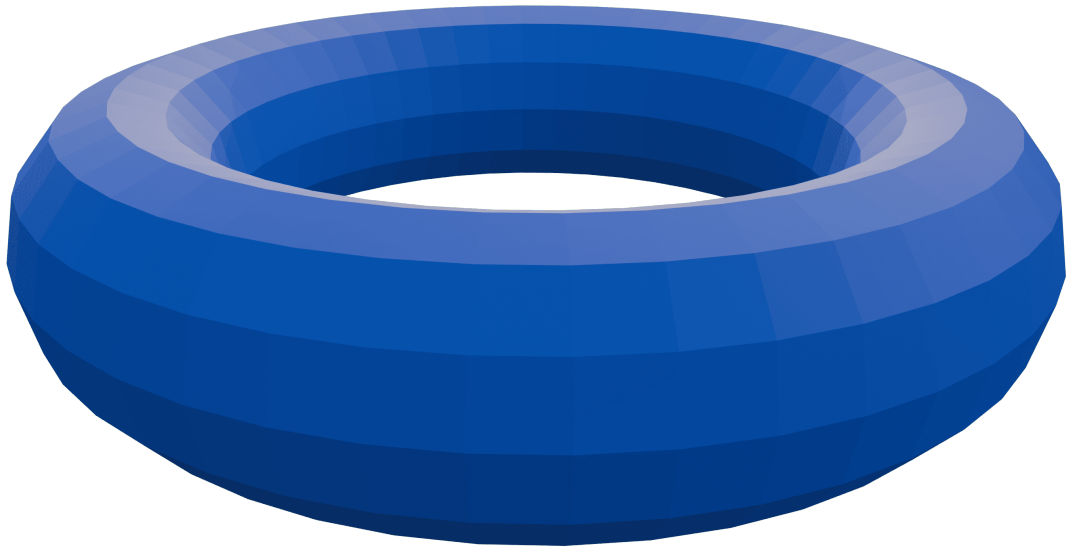
Takarási feladat

- Feladat: eldönteni, hogy a kép egyes részein milyen felületdarab látszik.
- Objektum tér algoritmusok:
  - Logikai egységenként dolgozunk, nem függ a képernyő felbontásától.
  - Rossz hír: általános esetben nem fog menni.
- Képtér algoritmusok:
  - Pixelenként döntjük el, hogy mi látszik.
  - Ilyen a sugárkövetés is.

Triviális hátlapeldobás, *Back-face culling*

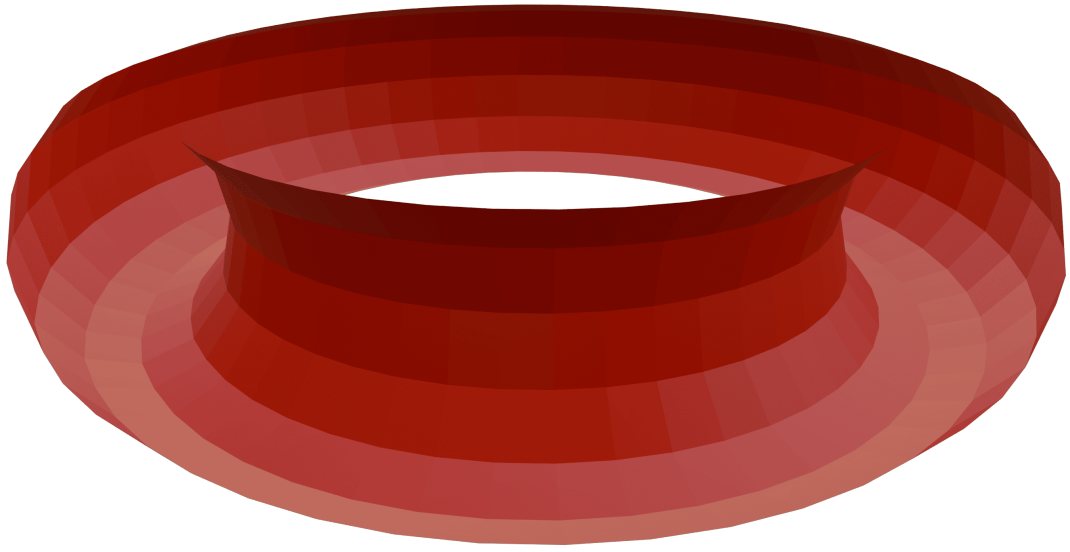
- Feltételezés: az objektumaink „zártak”, azaz ha nem vagyunk benne az objektumban, akkor sehonnan sem láthatjuk a felületét belülről.
- Körüljárási irány: rögzítsük, hogy a poligonok csúcsait milyen sorrendben *kell* megadni:
  - óramutató járásával megegyező (*clockwise, CW*)
  - óramutató járásával ellentétes (*counter clockwise, CCW*)
- Ha a transzformációk után a csúcsok sorrendje nem egyezik meg a megadással, akkor a lapot *hátulról* látjuk $\Rightarrow$ nem kell kirajzolni, *eldobható* – kb. a lapok fele.
- Ha csak egy objektumunk van és az konvex, akkor ennyi elég is – általában nem ez a helyzet.

Triviális hátlapeldobás



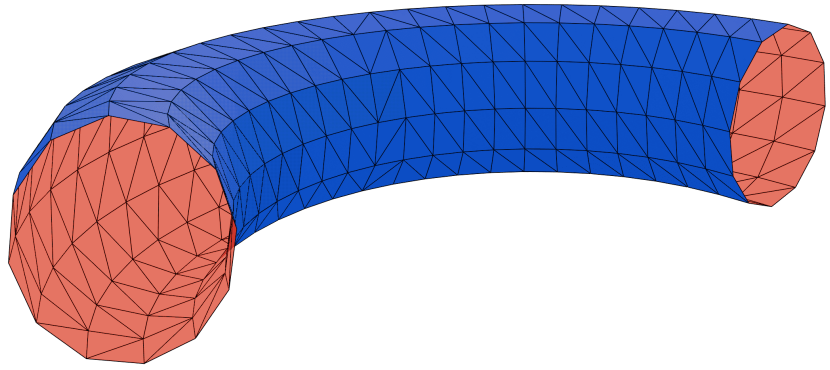
Előlapok megjelenítése

Triviális hátlapeldobás



Hátlapok megjelenítése – vagy ha a csúcsok sorrendjét felcseréljük

Triviális hátlapeldobás

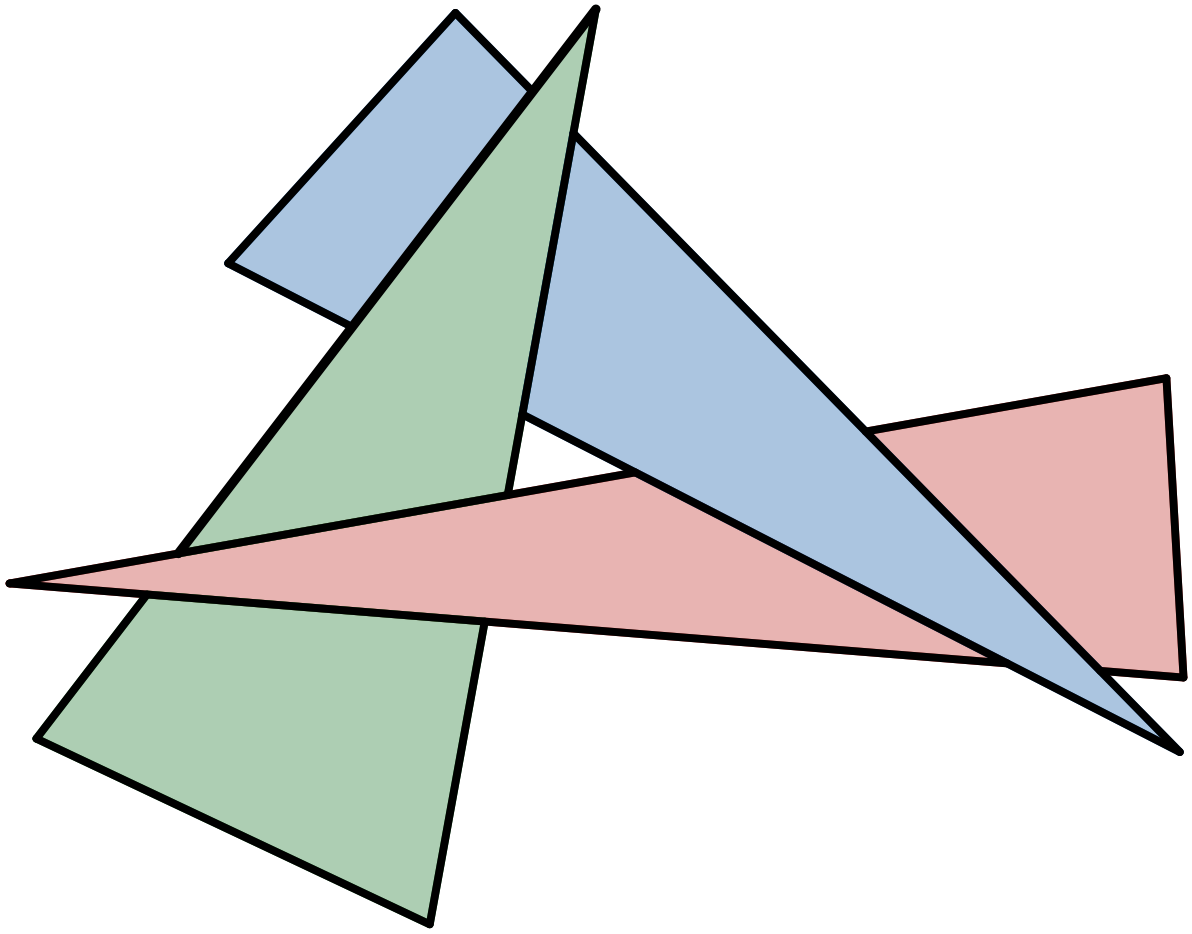


Ha az objektum nem zárt, szükség lehet az elő- és hátlapok megjelenítésére is

Festő algoritmus

- Rajzoljuk ki hátulról előre haladva a poligonokat!
- Ami közelebb van, azt később rajzoljuk $\Rightarrow$ ami távolabb van, takarva lesz.
- Egyszerűbb esetekre (pl. 2D, GUI) jó megoldás
- Probléma: hogyan rakjuk sorrendbe a poligonokat?
- Már néhány háromszögnél is előfordul olyan eset, amikor nem lehet egyértelmű sorrendet megadni.

Festő algoritmus – amikor nem működik



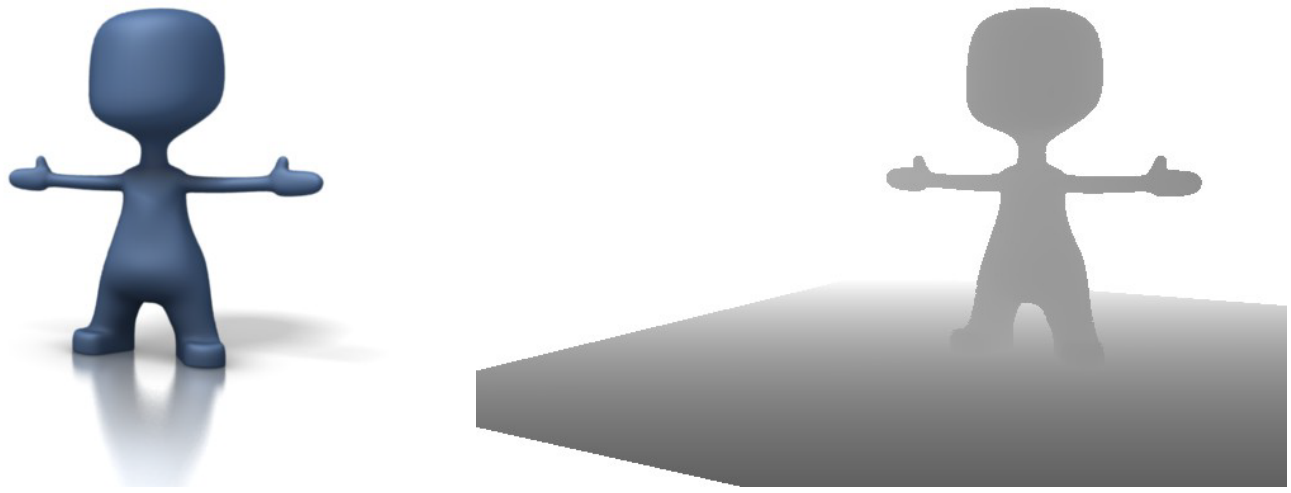
Z-buffer algoritmus

- Képtérbeli algoritmus
- A primitívek kirajzolási sorrendje nem számít
- Minden pixelre nyilvántartjuk, hogy ahhoz milyen mélységérték tartozott.
- Ha megint újra erre a pixelre rajzolnánk (*Z-teszt*):
  - **Ha** az új Z érték mélyebben van, **akkor** ez a pont takarva van $\Rightarrow$ nem rajzolunk
  - **Ha** a régi Z érték mélyebben van, **akkor** az új pont kitakarja azt $\Rightarrow$ rajzolunk *és* eltároljuk az új Z értéket.

Z-buffer

- Z-buffer vagy mélységi buffer (*depth buffer*): külön memóriaterület.
- Képernyő/ablak méretével megegyező méretű tömb.
- Minden pixelhez tartozik egy érték a bufferből.
- Ezzel kell összehasonlítani, és ide kell írni, ha a pixel *átment a Z-teszten*.
- Pontosság: a közeli és távoli vágósík közti relatív távolságtól függ. Kisebb pontosság könnyen előidézhet *Z-fighting*-ot.
- Gyakorlatban:
  - Hardveres gyorsítás, 16-32 bites elemek.
  - Pl. közeli vágósík: t, távoli: $1000t$, akkor a Z-buffer által tárolható értékek 95%-a a tartomány első 2%-át írja le.

Z-buffer



by macouno, macouno.com

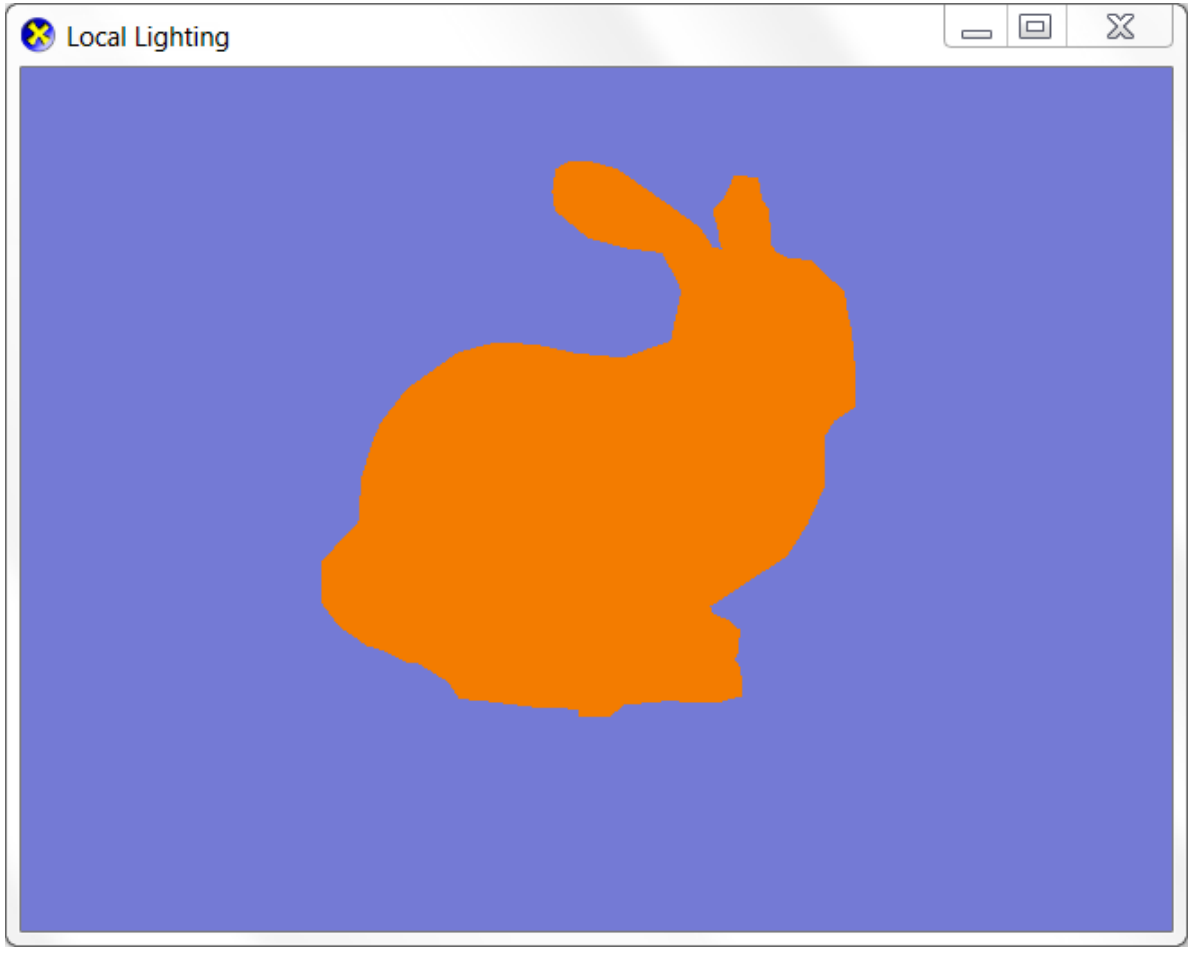
Lokális illumináció

- Ha már megvannak a primitívjeink pixelekre való leképezései, valahogyan számítsunk színeket

Saját színnel árnyalás

- Minden objektumhoz/primitívhez egy színt rendelünk, és kirajzoláskor ez lesz a pixelek értéke.
- Leggyorsabb: az illumináció gyakorlatilag egyetlen értékadás.
- Borzasztó: se nem valósághű, se nem szép.

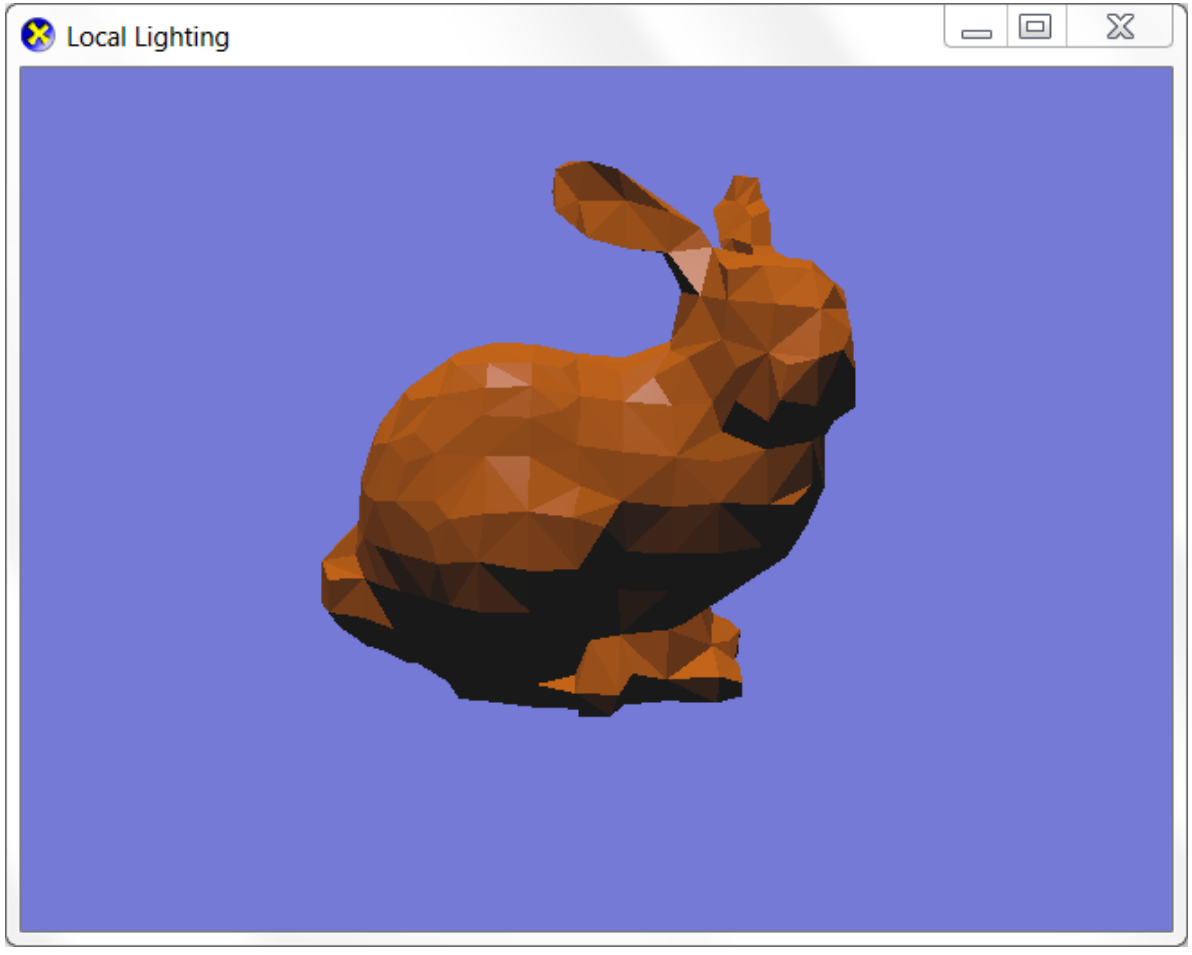
Saját színnel árnyalás



Konstans árnyalás, *Flat shading*

- A megvilágítást poligononként egyszer számítjuk ki, a szín homogén a lapon belül.
- Gyors: a műveletek száma a poligonok számától függ, a pixelek számától független.
- Van hogy használható: íves részeket nem tartalmazó, diffúz, egyszínű objektumokra.

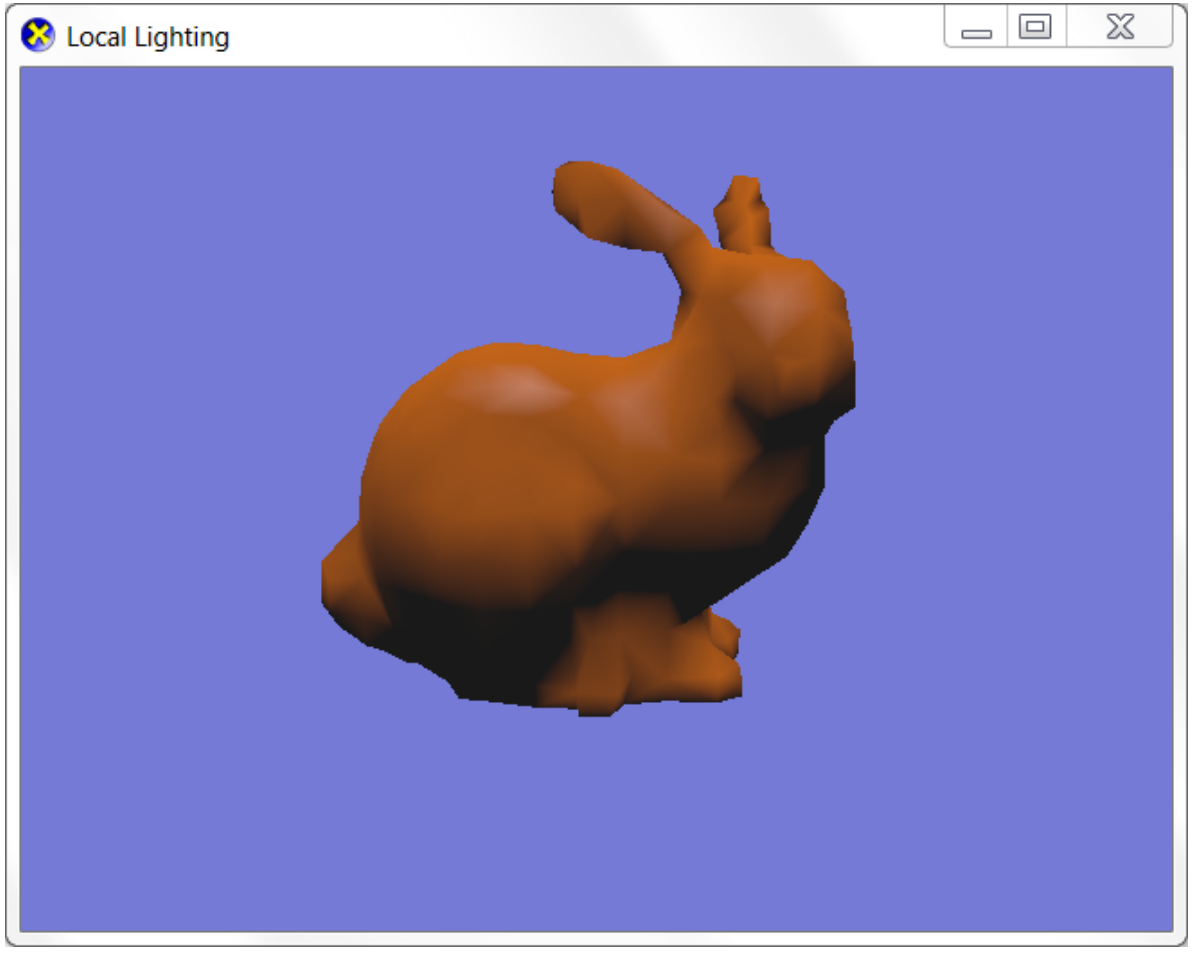
Konstans árnyalás, *Flat shading*



Gouraud-árnyalás

- A megvilágítást csúcspontonként számítjuk ki, a lapon interpolációval számítjuk a színeket a három csúcspontban kapott értékből.
- Lassabb: N db megvilágítás számítás + minden pixelre interpoláció.
- Szebb: az árnyalás minősége nagyban függ a poligonok számától. De nagy lapokon nem tud megjelenni a csillanás.
- Grafikus szerelőszalaghoz jól illik: a vertexek pozíciójának számítása mellett kiszámíthatjuk a színt is, utána a raszterizációnál automatikusan megkapják a létrejövő fragmentek az interpolált színt.

Gouraud-árnyalás



Phong-árnyalás

- Csak a normálvektorokat interpoláljuk, a megvilágítást minden pixelre kiszámítjuk.
- Leglassabb: *pixelek száma* db megvilágítás számítás.
- Legszebb: az árnyalás minősége nem függ a poligonok számától. Csillanás akár a poligon közepén is meg tud jelenni.
- Grafikus szerelőszalaghoz jól illik: a raszterizációnál a normálvektorokat interpoláltatjuk és minden fragmentre külön számítjuk a színt a kapott normálvektor segítségével.

Phong-árnyalás